

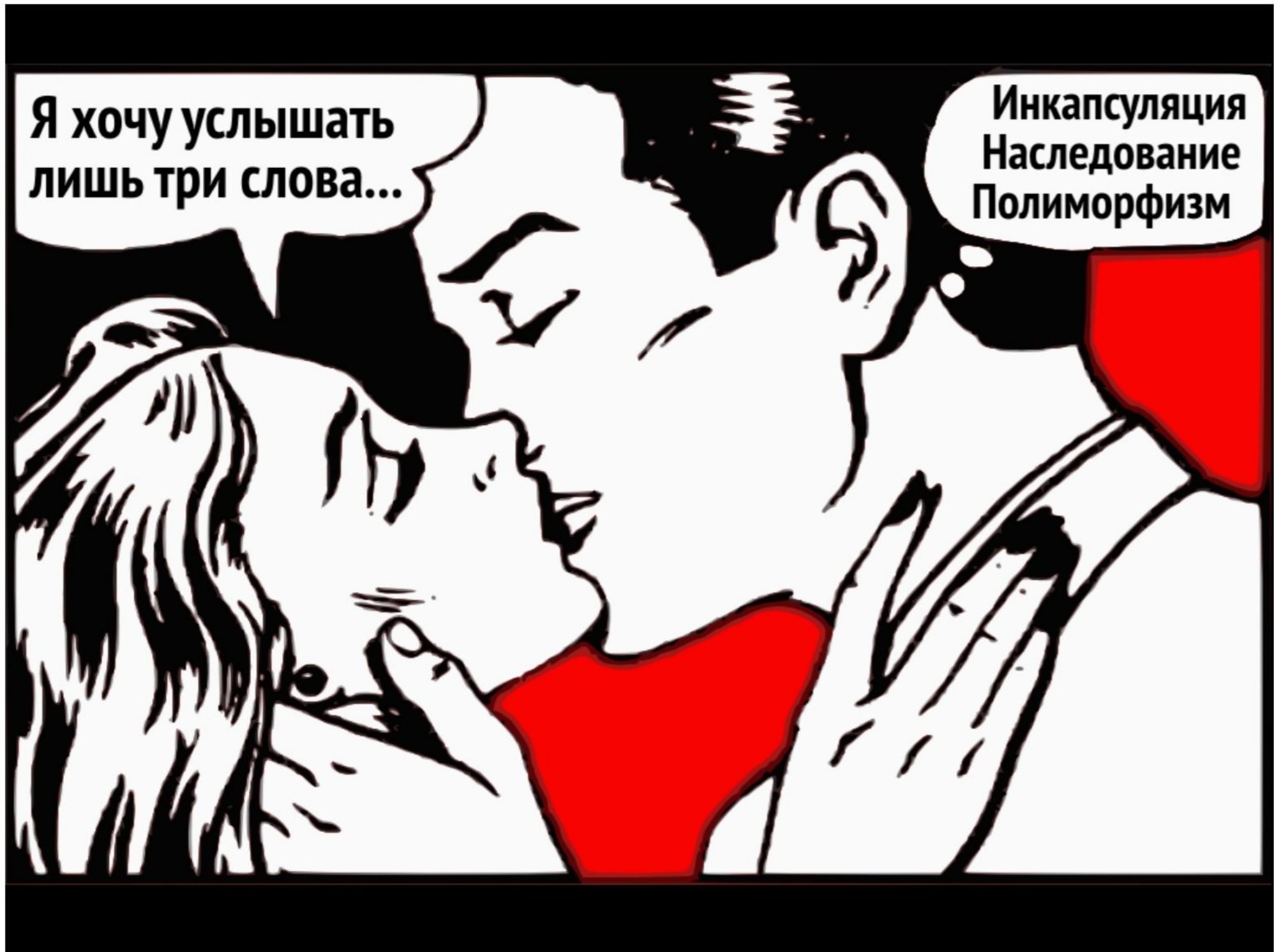


ООП

Часть II

Я хочу услышать
лишь три слова...

Инкапсуляция
Наследование
Полиморфизм



Инкапсуляция

По умолчанию атрибуты в классах являются общедоступными (`public`), а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его.

```
class Person:
    def __init__(self, name):
        self.name = name      # устанавливаем имя
        self.age = 1          # устанавливаем возраст

    def display_info(self):
        print(f"Имя: {self.name}\tВозраст: {self.age}")

obj = Person("Pupkin")
tom.age = 50                  # изменяем атрибут age
tom.display_info()           # Имя:Pupkin
                              Возраст: 50
```




Инкапсуляция

Все объекты в Python инкапсулируют внутри себя данные и методы работы с ними, предоставляя публичные интерфейсы для взаимодействия.

Атрибут может быть объявлен **приватным** (private) с помощью **нижнего подчеркивания** перед именем, но настоящего скрытия на самом деле не происходит – все на уровне соглашений.

```
class SomeClass:
    def _private(self):
        print("Это внутренний метод объекта")

obj = SomeClass()
obj._private() # это внутренний метод объекта
```



Два нижних подчеркивания (**double underscore**) - дандер

Если поставить перед именем атрибута два
подчеркивания, к нему нельзя будет обратиться
напрямую.

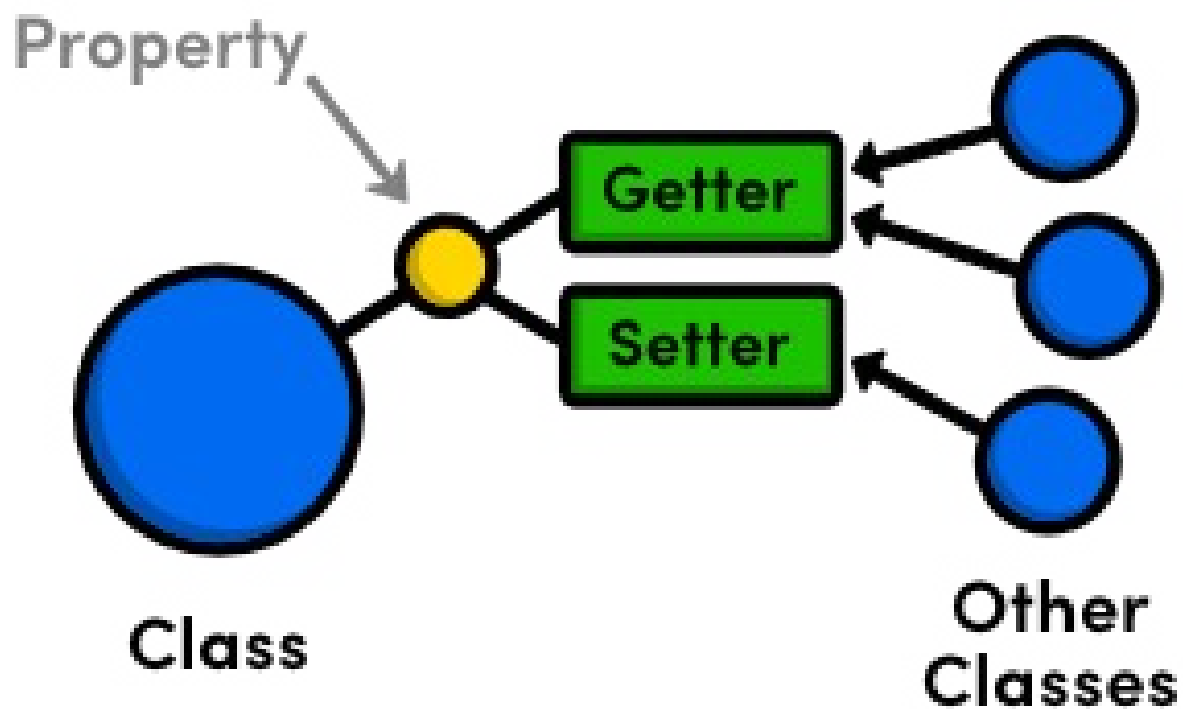
```
class SomeClass():  
    def __init__(self):  
        self.__param = 42 # приватный атрибут
```

```
obj = SomeClass()
```

```
obj.__param # AttributeError: 'SomeClass' object has  
no attribute '__param'
```

```
obj._SomeClass__param # - обходной способ
```

Методы доступа к свойствам



```
class Person:
    def __init__(self, name):
        self.__name = name # устанавливаем имя
        self.__age = 1 # устанавливаем возраст

    def set_age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

    def get_age(self):
        return self.__age

    def get_name(self):
        return self.__name

    def display_info(self):
        print(f"Имя: {self.__name}\tВозраст: {self.__age}")

tom = Person("Tom")
tom.display_info() # Имя: Tom  Возраст: 1
tom.set_age(25)
tom.display_info() # Имя: Tom  Возраст: 25
```



Встроенные методы

Вместо того чтобы вручную создавать геттеры и сеттеры для каждого атрибута, можно перегрузить встроенные методы

- `__getattr__`
- `__setattr__`
- `__delattr__`

Например, так можно перехватить обращение к свойствам и методам, которых в объекте не существует:



Встроенные методы `__getattr__`

автоматически вызывается при получении
несуществующего свойства класса

```
class SomeClass():  
    attr1 = 42  
  
    def __getattr__(self, attr):  
        return attr.upper()
```

```
obj = SomeClass()  
obj.attr1 # 42  
obj.attr2 # ATTR2
```



__getattr__

Перехватывает все обращения (в том числе и к существующим атрибутам):

```
class SomeClass():  
    attr1 = 42  
    def __getattr__(self, attr):  
        return attr.upper()  
  
obj = SomeClass()  
obj.attr1 # ATTR1  
obj.attr2 # ATTR2
```

Встроенные методы `__setattr__`

#автоматически вызывается при изменении свойства класса;

```
class SomeClass():  
    age = 42  
  
    def __setattr__(self, attr, value):  
        if attr == 'age':  
            print( 'Age, {} !'.format( value ) )  
            self.age = value
```

```
obj = SomeClass()  
obj.age # 45  
obj.age = 100 # Вызовет метод __setattr__
```

```
obj.name = 'Purkin'      ← Что произойдет при вызове ?
```



`__delattr__` удаление атрибута

```
class Car:
    def __init__(self):
        self.speed = 100
    def __delattr__(self, attr):
        self.speed = 42

# Создаем объект
porsche = Car()
print(porsche.speed)
# 100
delattr(porsche, 'speed') ← Удаление атрибута у объекта
print(porsche.speed) → 42
```

Перехват обращение к свойствам

Например, так можно перехватить обращение к свойствам и методам, которых в объекте не существует:

```
class AccessControl:
    def __setattr__(self, attr, value):
        if attr == 'age':
            self.__dict__[attr] = value
        else:
            raise AttributeError, attr + ' not allowed'
```

```
X = AccessControl()
X.age = 40
X.name = 'pythonlearn'
AttributeError: name not allowed
```

Если мы используем метод `__setattr__`, все присваивания в нем придется выполнять посредством словаря атрибутов. Используйте `self.__dict__['age'] = x`, а не `self.name = x`:

Декораторы свойств

Для создания свойства-геттера над свойством ставится аннотация **@property**.

Для создания свойства-сеттера над свойством устанавливается аннотация **имя_свойства_геттера.setter**.

```
class Person:
    def __init__(self, name):
        self.__age = 0 # устанавливаем возраст

    @property
    def age(self):
        return self.__age
    #Свойство-сеттер определяется после свойства-геттера.
    @age.setter
    def age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

tom = Person("Tom")
tom.age = -100 # Недопустимый возраст
```



Наследование

Одиночное наследование

#Родительский класс помещается в скобки после имени класса. Объект производного класса наследует все свойства родительского.

```
class Tree(object):
    def __init__(self, kind, height):
        self.kind = kind
        self.age = 0
        self.height = height
    def grow(self):
        """ Метод роста """
        self.age += 1

class FruitTree(Tree):
    def __init__(self, kind, height):
        # Необходимо вызвать метод инициализации родителя.
        super().__init__(kind, height)

    def give_fruits(self):
        print ("Collected 20kg of {}".format(self.kind))

f_tree = FruitTree("apple", 0.7)
f_tree.give_fruits()
f_tree.grow()
```




Множественное наследование

При множественном наследовании дочерний класс наследует все свойства родительских классов. Синтаксис множественного наследования очень похож на синтаксис обычного наследования.

```
class Horse():
    isHorse = True
class Donkey():
    isDonkey = True
class Mule(Horse, Donkey):
    pass
mule = Mule()
mule.isHorse # True
mule.isDonkey # True
```



Многоуровневое наследование

Мы также можем наследовать класс от уже наследуемого. Это называется многоуровневым наследованием. Оно может иметь сколько угодно уровней.

В многоуровневом наследовании свойства родительского класса и наследуемого от него класса передаются новому наследуемому классу.

```
class Horse():  
    isHorse = True  
class Donkey(Horse):  
    isDonkey = True  
class Mule(Donkey):  
    pass  
mule = Mule()  
mule.isHorse # True  
mule.isDonkey # True
```



Композиция

Где класс является один класс является полем другого.

```
class Salary:
    def __init__(self, pay):
        self.pay = pay

    def getTotal(self):
        return (self.pay*12)

class Employee:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus
        self.salary = Salary(self.pay)

    def annualSalary(self):
        return "Total: " + str(self.salary.getTotal()) +
self.bonus)

employee = Employee(100, 10)
print(employee.annualSalary())
```




Полиморфизм

Все методы в языке изначально **виртуальные**. Это значит, что дочерние классы могут их переопределять и решать одну и ту же задачу разными путями, а конкретная реализация будет выбрана только во время исполнения программы. Такие классы называют полиморфными.

```
class Mammal:
    def move(self):
        print('Двигается')
```

```
class Hare(Mammal):
    def move(self):
        print('Прыгает')
```

```
animal = Mammal()
animal.move() # Двигается
hare = Hare()
hare.move() # Прыгает
```



```
class Animal:
    def __init__(self, name):
        self.name = name
```

```
class Horse(Animal):
    def run(self, distance):
        print(self.name, "run", distance, "meters")

    def sound(self):
        print(self.name, "says: igogo")
```

```
class Bird(Animal):
    def fly(self, distance):
        print(self.name, "fly", distance, "meters")

    def sound(self):
        print(self.name, "says: chirik")
```

```
class Unicorn( Bird, Horse):
    pass
```

```
pegas = Unicorn("Пегас")    ← Вызов конструктора баз. класса
pegas.run(20)
pegas.fly(800)
pegas.sound()               ← Что будет издавать Пегас ?
```



Продолжение следует ...